View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Elsevier - Publisher Connector

Electronic Notes in Theoretical Computer Science 65 No. 7 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume65.html 14 pages

Specification and Verification using Message Sequence Charts

Doron Peled

Department of Electrical and Computer Engineering
The University of Texas
Austin, TX 78712, USA

Abstract

Message sequence charts is a notation used in practice by protocol designers and system engineers. In this survey, some of the recent results related to this notation, in the context of specification and automatic verification of communication protocols, are presented.

1 Introduction

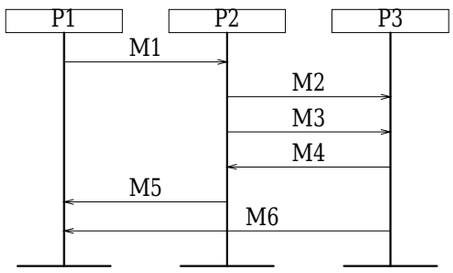
Specifying the behavior of software systems is of major interest for engineers. When concurrency is involved, things become even more challenging. Even before considering the actual notation to be used for specification, there is a large choice of models of execution. Different models vary in the detailed information they carry, the intuition they provide and the difficulty of checking properties of the modeled systems.

After the selection of the model, we are still left with a wide choice of notation, affecting our level of abstraction and the complexity of deciding their properties. Perhaps the most successful model is that of state machines. This model enjoyns several desirable properties. An interleaved execution is simply a sequence or a path in the graph of the state machine. Linear structures are easy to work with. Simple formalisms, such as linear temporal logic are available. In the finite case, there are simple decision procedures for checking properties of such models.

Message sequence charts (MSCs) is a partial-order based standard [6] formalism. It has a visual notation, which clearly demonstrates the interaction between the involved concurrent processes. It is already used in practice by protocol designers, a fact which gives it an advantage over other formalisms in technology transfer. On the other hand, working with an existing standard, which was developed by a

[1] Email:doron@ece.utexas.edu

©2002 Published by Elsevier Science B. V. Open access under CC BY-NC-ND license.




```
msc MSC;
inst P1: process Root,
P2: process Root,
P3: process Root;
instance P1;
out M1 to P2;
in M5 from P2;
in M6 from P3;
endinstance;
instance P2;
in M1 from P1;
out M2 to P3;
out M3 to P3;
in M4 from P3;
out M5 to P1;
endinstance;
instance P3;
in M2 from P2;
in M3 from P2;
out M4 to P2;
out M6 to P1;
endinstance;
endmsc;
```

Fig. 1. Visual and textual representation of an MSC

committee, without a full view of algorithms and complexity issues, can be challenging and, on the other hand, restrictive. In this survey we discuss several issues in specification and verification using message sequence charts.

2 Preliminaries

Each MSC describes a scenario where some processes communicate with one another. Such a scenario includes a description of the messages sent, messages received, the local events, and the ordering between them. In the visual description of MSCs, each process is represented as a vertical line, while a message is represented by a horizontal or slanted arrow from the sending process to the receiving one, as appears in the left part of Figure 1. The corresponding textual representation of the MSC appears in the right part of Figure 1.

Definition 2.1 An MSC M is given as a tuple $\langle V, <, \mathcal{P}, \mathcal{N}, L, T, N, m \rangle$, where

- V is a finite set of events,
- $< \subseteq V \times V$ is an acyclic relation,
- $\mathcal{P}$ is a set of processes,
- $\mathcal{N}$ is a set of *message names*,
- $L : V \to \mathcal{P}$ is a mapping that associates each event with a process,
- $K : V \to \{\mathrm{s}, \mathrm{r}, \mathrm{l}\}$ is a mapping that describes the *kind of* each event as *send*, *receive* or *local*, respectively,
- $N : V \to \mathcal{N}$ maps every event to a name,
- $m \subseteq V \times V$ is relation called *matching* that pairs up *send* and *receive* events. Each *send* is paired up with exactly one *receive* and vice versa. Events v_1 and v_2

can be paired up with each other, only if $N(v_1) = N(v_2)$.

A *type* is a triple (i, j, C), including the indexes of the sending process $P_i \in \mathcal{P}$ and receiving process $P_j \in \mathcal{P}$, and a message name $C \in \mathcal{N}$. Each *send* or *receive* event has a type, according to the origin and destination of the message, and the label of the message. The type of a local event of process $P_i \in \mathcal{P}$ is (i, i). Matching events have the same type. A message consists of a pair of matched *send* and *receive* events. For two events v_1 and v_2, we have $v_1 < v_2$ if and only if one of the following holds:

- v_1 and v_2 are matching *send* and *receive* events, respectively.
- v_1 and v_2 belong to the same process, with v_1 appearing before v_2 on the process line.

We assume FIFO (first in first out) message passing, i.e.,

$$(K(v_1) = K(v_2) = \mathrm{s} \wedge K(v_1') = K(v_2') = \mathrm{r} \wedge\ m(v_1, v_1') \wedge m(v_2, v_2') \wedge$$
$$L(v_1) = L(v_2) \wedge L(v_1') = L(v_2') \wedge v_1 < v_2) \Rightarrow v_1' < v_2'$$

Denote by $u \longrightarrow v$ the fact that $u < v$ and either u and v are matching *send* and *receive* events, or u and v belong to the same process and there is no event between u and v on the same process line. That is, u immediately precedes v. The transitive closure $<^*$ of the relation $<$ is a partial order called the *visual ordering* of events. Clearly, the visual ordering can be defined equivalently as the transitive closure of the relation $\longrightarrow$. The MSC notation represents a partial order execution, where the fact that two events u, v are ordered according to the visual order means that u happens before v. A *linearization* of an MSC $M = \langle V, <, \mathcal{P}, \mathcal{N}, L, K, N, m\rangle$ is a total order on V, which extends the relation $(V, <)$.

Example 2.2 Consider the example MSC given in Figure 1. Denote by v_i the *send* event and by w_i the receive event of message Mi, $1 \leq i \leq 6$. Then we have $V = \{v_1, \ldots, v_6, w_1, \ldots, w_6\}$, $\mathcal{P} = \{P1, P2, P3\}$, $\mathcal{N} = \{M1, \ldots, M6\}$ and $N(v_i) = N(w_i) = Mi$ for all i. The events located on $P1$ are $L^{-1}(P1) = \{v_1, w_5, w_6\}$, with $K(v_1) = \mathrm{s}$, $K(w_5) = K(w_6) = \mathrm{r}$, and $v_1 < w_5 < w_6$. This ordering is the time ordering of events on $P1$. We also have $m(v_i, w_i)$ and $v_i < w_i$ for all i (message ordering). In particular, $v_1 < w_1 < v_2 < w_2$.

The partial order between the *send* and *receive* events of Figure 1 is shown in Figure 2. In this figure, only the 'immediately precedes' order $\longrightarrow$ is shown. Notice for example that the *send* events v_5 and v_6, of the two messages, $M5$ and $M6$, respectively, are unordered.

Definition 2.3 The *concatenation* $M_1 M_2$ of two MSCs,

$$M_k = \langle V_k, <_k, \mathcal{P}, \mathcal{N}_k, L_k, K_k, N_k, m_k\rangle, \text{ for } k = 1, 2.$$

over the same set of processes $\mathcal{P}$ and disjoint sets of events $V_1 \cap V_2 = \emptyset$ (we can always rename events so that the sets become disjoint) is defined as $\langle V_1 \cup V_2, <,$

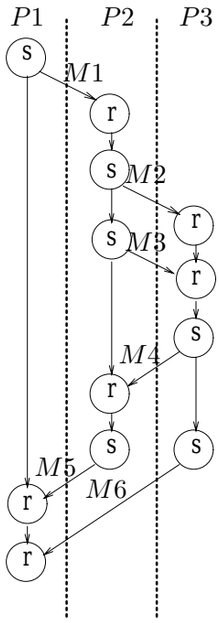


Fig. 2. The partial order between the events of the MSC in Figure 1.

$\mathcal{P}, \mathcal{N}_1 \cup \mathcal{N}_2, L_1 \cup L_2, K_1 \cup K_2, N_1 \cup N_2, m_1 \cup m_2\rangle$, where

$$< = <_1 \cup <_2 \cup \{(u,v) \in V_1 \times V_2 \mid L_1(u) = L_2(v)\}\,.$$

That is, the events of M_1 precede the events of M_2 for each process, respectively (but some events in M_1 of one process may be unordered with respect to some events in M_2 of another process). If $M = M_1\, M_2$, we say that M_1 is a *prefix* of M, and denote this by $M_1 \sqsubseteq M$ (this also means containment between the different process events of the MSCs M_1 and M). Notice that no synchronization of the different processes is assumed in the definition of concatenation. Thus, $M_1\, M_2$ does not necessarily describe a behavior that starts according to M_1 and after completing all the events from M_1 progresses to behave according to the events in M_2. In particular, it is possible in $M_1\, M_2$ that one process is still involved in some actions of $M1$, while another process has advanced to events from $M2$. The infinite concatenation of finite MSCs is defined in a similar way, and it allows defining infinite MSCs as well.

Definition 2.4 Let $M_1, M_2, \ldots$ be an infinite sequence of finite MSCs. Define a sequence $M_1{}', M_2{}', \ldots$ as follows: Let $M_1{}' = M_1$, and for $i > 1$, $M_i{}' = M'_{i-1} M_i$. (Thus, if $i < j$, $M_i{}' \sqsubseteq M_j{}'$.)

Let $M'_i = \langle V_i, <_i, \mathcal{P}, \mathcal{N}_i, L_i, K_i, N_i, m_i\rangle$. Then the infinite concatenation $M_1 M_2 \ldots$ is defined as the infinite MSC $M = \langle V, <, \mathcal{P}, \mathcal{N}, L, K, N, m\rangle$ where $V = \cup_{i\geq 1} V_i$ is the disjoint union of the V_i, $\mathcal{N} = \cup_{i\geq 1}\mathcal{N}_i$, $L|_{V_i} = L_i$, $K|_{V_i} = K_i$, $N|_{V_i} = N_i$ ($K|_{V_i}$ and $N|_{V_i}$ denote the functions K and N, respectively, restricted to the domain V_i), $m = \cup_{i\geq 1} m_i$ and

$$< = \cup_{i\geq 1} <_i \ \cup \{(u,v) \mid L_i(u) = L_j(v) \wedge u \in V_i \wedge v \in V_j \wedge i < j\}\,.$$

(Note that in the last line, the first use of '$<$' refers to the relation between events that is defined here, while the last '$<$' is the usual 'smaller than' relation between natural numbers.)

Since a communication system usually includes many (or even infinitely many) such scenarios, a high level description is needed for combining them together. The standard description consists of a graph called HMSC (high-level MSC), where each node contains one MSC as in Figure 3. Each maximal path in this graph (i.e., a path that is either infinite or ends with a node without outgoing edges) that starts from a designated initial state corresponds to a single *execution* or *scenario*.

Definition 2.5 An HMSC is a 4-tuple $\langle \mathcal{S}, \mathcal{M}, c, \tau, \mathcal{S}_0 \rangle$ where $\mathcal{S}$ is a finite set of nodes, $\mathcal{M}$ is a set of finite MSCs with sets of events disjoint from one another. The mapping $c : \mathcal{S} \to \mathcal{M}$ associates the node g with an MSC $c(g)$. By $\tau \subseteq \mathcal{S} \times \mathcal{S}$ we denote the *edge relation*. The *initial* nodes $\mathcal{S}_0$ are a subset of $\mathcal{S}$. An *execution* of the HMSC is a (finite or infinite) MSC $c(g_0)\, c(g_1)\, c(g_2)\, \cdots$ associated with a maximal path $g_0, g_1, \ldots$ of the HMSC that starts with some initial node $g_0 \in \mathcal{S}_0$.

Figure 3 shows an example of an HMSC where the node in the upper left corner is the starting node. Initially, process P_1 sends a message to P_2, requesting a connection (e.g., to an internet service), according to the top left box. This can result in either an approval message from P_2, according to the top right box, or a failure message, according to the bottom left box. In the latter case, a report message is also sent from P_2 to some supervisory process P_3. There are two progress choices, corresponding to the two arrows out of the bottom left box. We can decide to try and connect again, by choosing the up arrow, or to give up and send a service request (from process P_1 to process P_3), by choosing the left-to-right arrow. Note how the HMSC description abstracts away from internal process computation, and presents only the communications. The executions of this system are either finite or infinite. Note further that according to HMSC semantics, process $P2$ in Figure 3 does not necessarily have to send its Report message before the execution of process $P1$ has progressed into the next node and has sent its Req_service message. However, process P3 must receive the Report message before the Req_service message.

According to the ITU standard [6], an HMSC can be hierarchical, i.e., an HMSC node can be mapped into another (lower level) HMSC. We ignore this feature, which is orthogonal to the discussion in this survey.

3 Expressiveness

Message sequence charts (MSCs) (including the extension to High level MSCs, i.e., HMSCs) is a formalism that is actively used in practice by protocol developers and software engineers. Unlike some other specification formalisms, it was not designed by researchers to fit into existing tools. This calls for studying of its properties, in an attempt to adapt some formal methods techniques, or develop new ones.

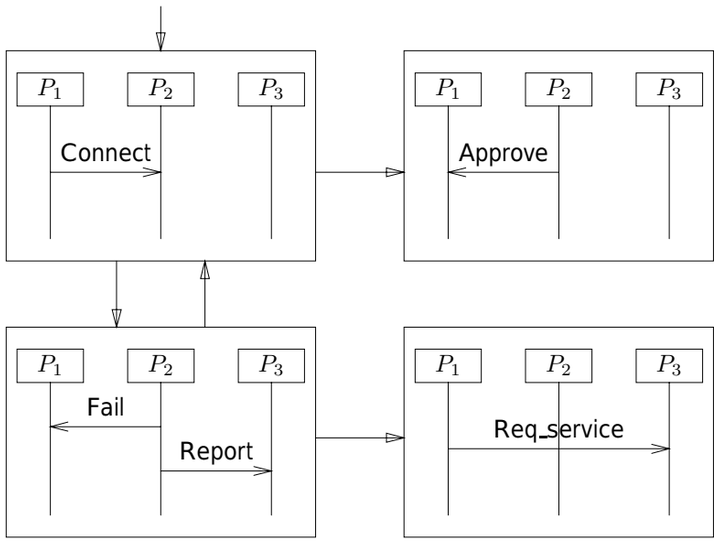


Fig. 3. An HMSC graph

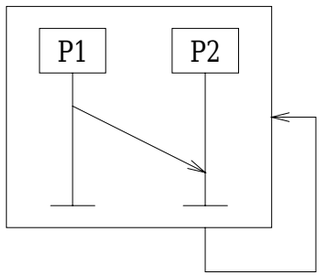


Fig. 4. Simple example with infinite state space

There are several interesting aspects of the MSC notation that pose a challenge to the researchers and the developers of tools. For example, the HMSC notation does not necessarily represent finite state systems, as there is no bound on the size of message queues. This fact has implications on the ability to automatically verify properties of HMSCs.

Consider for example the HMSC in Figure 4. This is the simplest example of an HMSC with infinitely many global states. In order to formalize this observation, we define the notion of a *global state* of an MSC.

Definition 3.1 Let $M = \langle V, <, \mathcal{P}, \mathcal{N}, L, T, N, m \rangle$ be a finite or infinite MSC (the latter case is obtained, e.g., by an infinite execution of an HMSC). Recall that $<^*$ is the transitive closure of $<$. A *global state* G is a finite subset of the events of V, such that if $f \in G$ and $e <^* f$, then $e \in G$. (We say that G is 'history closed'.)

A global state is usually defined as an assignment function from the program variables to their values. In the MSC context, the assignment can return the number of events in the different message queues. This number is obtained by counting for

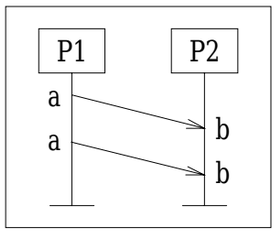


Fig. 5. An MSC with two messages

each pair of processes the number of *send*s and substructing from it the number of *receive*s. Now, it is easy to see that the states of the unique and infinite execution of the HMSC in Figure 4 consist of m *send*s and n *receive*s for any natural $m \geq n$.

It is interesting to know what is the expressive power of HMSCs. In order to remain within the domain of formal languages, we will look at the *linearizations* of MSC executions, i.e., their completions into total orders. We will label each event in an MSC node with a label from a finite alphabet Σ. We allow (but do not force) labeling of different events of the *same type and kind* by the same letter.

Consider the MSC in Figure 5. It has two messages, i.e., 4 events. We labeled the *send*s with as, and the *receive*s with bs. This MSC generates two linearizations (words): $abab$ and $aabb$. These languages of linearizations are closed under certain permutation of adjacent occurrences of events. We have three *permutation rules*:

(i) If a is a *receive* from P_i to P_j, and b is a *send* from P_i to P_j, then we can permute $\sigma_i ab\sigma_2$ ($\sigma_1, \sigma_2 \in \Sigma^*$) to obtain $\sigma_2 ba\sigma_2$. Note this rule does not permit us to to permute in the reverse direction, i.e., from $\sigma_1 ba\sigma_2$ to $\sigma_1 ab\sigma_2$.

(ii) If a is a *send* from P_i to P_j, and b is a *receive* from P_i to P_j, we can permute a with b in $\sigma_1 ab\sigma_2$ provided that the following condition hold: $\#_a\sigma_1 > \#_b\sigma_1$, where $\#_c\sigma$ is the number of cs appearing in the word σ.

(iii) If a and b belong to different processes, and their types do not match as in the previous case, then we can permute a with b. (In fact, we can also permute b with a, from the symmetry of this condition.)

The reverse permutation of the first rule may allow a *receive* to appear before the corresponding *send*. For example, given the linearization $abab$ of the MSC in Figure 5, we cannot permute the first a with the first b to obtain $baab$. The second rule specifies the condition under which the reverse permutation is allowed. Under this rule, the adjacent a and b, which can be permuted, are *not* a matching pair. Also note that for MSCs, it is not possible to use a fixed independence relation between events, as in *trace theory* [9].

We can define the language of an HMSC as follows:

(i) Let $\mathcal{L}(M)$ be the finite language of an HMSC node M.

(ii) Let $\mathcal{K}$ be the language of the graph of the HMSC, where each node in the graph is assigned some unique letter (disjoint from the letters in Σ). According to Kleene's construction, the language $\mathcal{K}$ is a regular language.

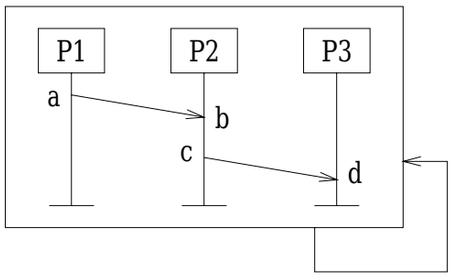


Fig. 6. An MSC with context-free behavior

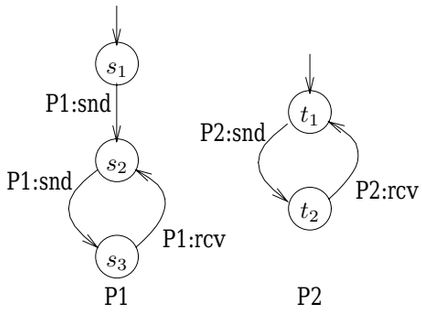


Fig. 7. A simple two process protocol

(iii) Substitute in $\mathcal{K}$ each letter corresponding to a node M by the language of $\mathcal{L}(M)$. This is still a regular language. We obtain a language $\tilde{\mathcal{K}}$.

(iv) Now close $\tilde{\mathcal{K}}$ under the permutation rules to obtain $[\tilde{\mathcal{K}}]$. Such permutations are achieved by using context sensitive grammar rules of the form $XabY \rightarrow XbaY$. Hence the language $[\tilde{\mathcal{K}}]$ of an HMSC is context sensitive. Note that we only permute events according to the first and third permutation rules given above. This is sufficient due to the fact that we took *all* the linearizations of each separate MSC node.

We saw that HMSC languages are obtainable from regular languages (ω-regular in the case of infinite executions) by closing under a given set of permutations.

To show that the language of HMSCs is, in general, not regular or context free, consider the example in Figure 6. The global states of this example have l times a events, m times c events, and n times d events, where $l \geq m \geq n$ (also the number of b events is the same or greater than, by exactly one, than the number of c events). This can be easily shown not to be in the class of context-free (and hence also not regular) languages.

We saw that an HMSC can represent a language that is not necessarily regular, in fact, not even context free. On the other hand, the MSC graph notation of the standard does not allow representing all the possible communication skeletons of finite state communication protocols. This makes HMSCs uncomparable with regular languages.

As an example, consider the infinite MSC that is generated from the simple

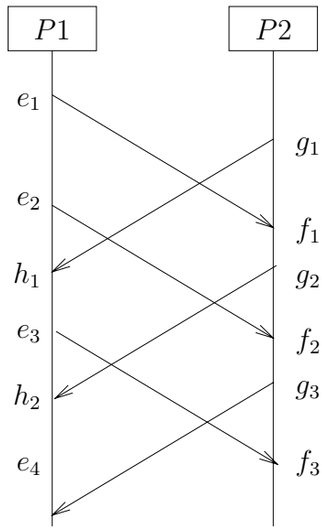


Fig. 8. A prefix of an MSC execution that cannot be decomposed

protocol in Figure 7. A finite prefix of the MSC description of the (unique and infinite) execution of this protocol appears in Figure 8. We show that this infinite MSC cannot be decomposed into a concatenation of finite MSCs. We start with the *send* event e_1 and *receive* event f_1. Obviously, because of the compulsory matching in HMSCs, they must belong to the same MSC node. We have the *send* event g_1 preceding f_1, on the same process line, while its corresponding *receive* event h_1 succeeds the *send* e_1. Thus, the events g_1 and h_1 must be in the same HMSC node with e_1 and f_1. For the same reason, we have that e_2 and f_2 must belong to the same node with g_1, and h_1, and so forth.

The problem lies within the restriction of the MSC nodes to contain matched messages. A different view of the expressiveness problem is that any global state that corresponds to a finite path in an HMSC (i.e., a global state that contains complete MSC nodes) has a matched set of *send* and *receive* events. In the partial order execution in Figure 8, there is no global state with this property. Hence, we cannot decompose this execution into finite MSCs (which will occur infinitely many times along some path of an HMSC).

An extension of the HMSC notation is described in [4]. It allows MSC nodes with unmatched *send* and *receive* events. Thus, a *send* event in one node may be matched with a *receive* event in a later node.

4 Undecidable Versions of Model Checking for HMSCs

Once we characterize HMSC languages as context sensitive languages, it is not too surprising that certain decision problems become undecidable. In particular, it is known that the emptyness of the intersection of two context sensitive languages is undecidable. We still have to prove that for HMSC languages, as they form a subset

of the context sensitive languages.

For a practical motivation of this, consider the common model checking approach, where we describe the *bad execution sequences* using a formalism we use for modeling the system (usually, finite automata over infinite words) [5,7,14]. The intersection of the executions contains bad sequences that are allowed by the checked system, which need to be reported. We can try, along these lines, to specify the bad or unwanted executions of a system using the HMSC formalism. If the intersection of the linearizations of two HMSCs is nonempty, we can easily take one and generate back an MSC. Intersecting two HMSCs is undecidable, as we show below [11]:

Theorem 4.1 *The problems of intersection of two HMSCs is undecidable.*

Proof. By reduction from Post Correspondence Problem (PCP). The input for PCP is a finite sequence of pairs of words

$$(w_1, v_1), (w_2, v_2), \ldots, (w_n, v_n)$$

The problem is to decide whether there is a finite sequence of indexes $i_1, i_2, \ldots, i_m$ such that $w_{i_1} w_{i_2} \ldots w_{i_m} = v_{i_1} v_{i_2} \ldots v_{i_m}$.

We construct two HMSCs. One for concatenating words that appear in the left components of the above pairs, and one for concatenating words that appear in the right components. Consider the HMSC for the left components. We have 4 processes $P1, \ldots P4$. For each word w_j, we construct an MSC node M_j with messages from $P1$ to $P2$ labeled by the letters of w_j. We have a node R_j, with one message, from $P3$ to $P4$, labeled by the index j. We also have an initial node E, with a message from $P1$ to $P4$, and a node F, with a message from $P4$ to $P1$. The structure of the automaton can be represented by the regular expression $E(+_{j=1..n} M_j R_j)^+ F$, which is also demonstrated in Figure 9.

The automaton for concatenating the right components is constructed similarly. Now notice that the events in the M_j components can commute with the events in the R_j components, since they involve disjoint processes. Therefore, any word in the intersection has the same characters according to the sequence of M_js, and the same indexes according to the sequence of R_js. ∎

Another attempt for providing model checking is to write the specification (or the negation of the specification, describing the bad executions) using an automaton over finite or infinite words, or using linear temporal logic (LTL). Unfortunately, the intersection of HMSC languages with regular languages, or the language of words satisfying linear temporal logic formulas, is undecidable as well.

To see that, replace in the previous proof the HMSC for the right subwords, the 'specification automaton', by an LTL (or regular expression, or a finite state automaton over infinite words) that represents some of the linearizations of the HMSC as follows: for an MSC node M, let $lin(M)$ be the single linearization of M that includes matching *send* and *receive* events appearing adjacent. (Note that his kind of linearization is not always possible for an MSC, see e.g., Figure 8. But is possible in our case because of the particular construction of the nodes in the reduction.) Thus the linearization of M_j, representing the word $w_j = \alpha\beta\beta\alpha$ will

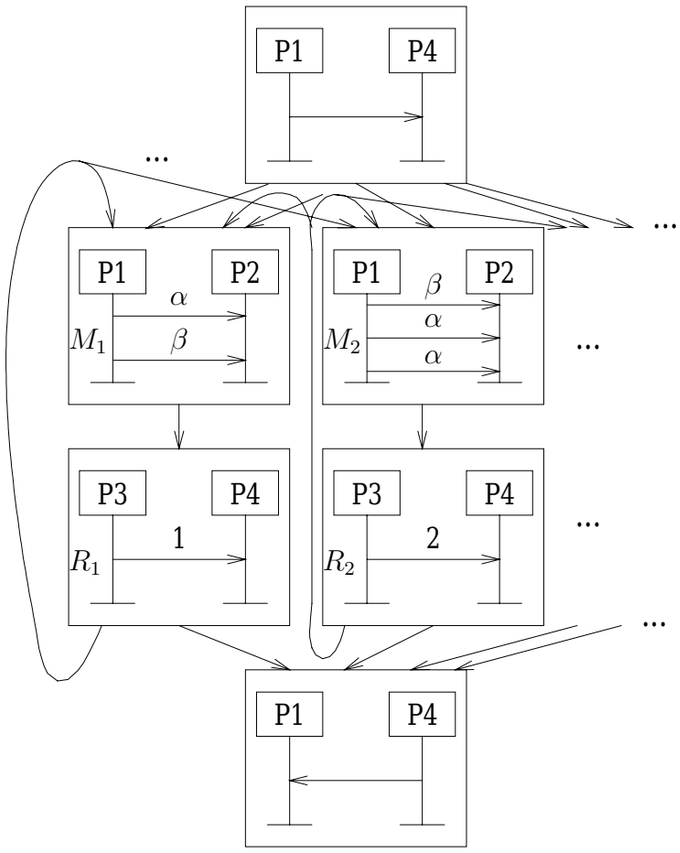


Fig. 9. An HMSC graph for the PCP reduction

be $s_\alpha r_\alpha s_\beta r_\beta s_\beta r_\beta s_\alpha r_\alpha$, where s_ρ represents a *send* of a message labeled by ρ, and r_ρ represents a *receive* of that message. The LTL formula will represent the language $lin(E)(+_{j=1..n} lin(M_j)lin(R_j))^+ lin(F)$ (this is a couner-free language, and thus can be represented using LTL).

The intersection of the (language of the) HMSC, representing the left words in the PCP problem, and the language of the LTL formula above, representing the right words, would include exactly the words that are solutions to the PCP problem. Hence, LTL model checking of HMSCs is undecidable.

5 Decidable Versions of Model Checking for HMSCs

There are several positive solutions for providing model checking algorithms for HMSCs.

Limiting the MSC. One possibility is to limit the message queue to some fixed size. In this case, HMSCs become finite state systems, and the usual model checking approaches can be used. Another constraint is the following [3,10]: the communication graph of an MSC M contains the processes as nodes, and an

edge from Pi to Pj if there is a communication from Pi to Pj in M. M has *bounded communication*, if its communication graph has a strongly connected component. The above model checking approaches become decidable if we require that every cycle in the HMSC has bounded communication. In this case, the HMSC language is regular. This result is also related to the star problem in trace languages [12]. Checking for bounded communication is NP-hard in the size of the HMSCs [10].

Allowing 'gaps' in the semantics of the specification HMSC. The HMSC representing the bad executions is interpreted in a different way than the HMSC representing the system. The former represents only part of the events. In particular, two adjacent events a and b on the same process line of the specification HMSC may match some nonadjacent events of the same type in the system HMSC. The pattern matching problem between these two HMSCs is decidable, and is in NP-hard, in the size of the HMSCs [11].

Using a partial order based specification formalisms. Consider a specification that has a language $\mathcal{L}$ that is regular and is already closed under the permutation rules. The emptiness of the intersection of such a specification with an HMSC language can be decided. The reason is that an HMSC language $[P]$ is generated from a regular language P by closing it under permutations. If $\mathcal{L} = [\mathcal{L}]$, then $\mathcal{L} \cap P \neq \emptyset$ iff $\mathcal{L} \cap [P] \neq \emptyset$. Thus, it is sufficient to check the emptiness of the intersection of $\mathcal{L}$ with the regular generator P of the HMSC language.

A solution that involves partial order based formalisms is the use of a subset of the logic TLC [2], as applied on HMSCs in [13]. According to this solution, we use temporal modalities to reason over the events of the MSC system. We use the same modalities symbols as in LTL, but give them a different interpretation; over paths of events, generated by the $<$ relation, rather than over linearizations of the partial order.

Thus, the assertion $\bigcirc\varphi$ holds for events that have an immediate successor under the relation $<$ for which φ holds. $\Diamond\varphi$ holds for events e from which there is a path according to $<$, leading to some event f for which φ holds (thus, $e <^* f$). Similarly, for $\psi\mathcal{U}\varphi$ to hold for e, we require, that ψ holds for each event along such a path from e to some event f where φ holds. Finally, in order to satisfy the usual duality $\Box\varphi = \neg\Diamond\neg\varphi$, we interpret $\Box\varphi$ as follows: it holds for events e that satisfy that for every event f such that $e <^* f$, φ holds for f.

Another decidable model checking solution is based on using second order monadic logic over partial orders [8].

6 Other Decision Problems

A natural problem that arises with MSCs is whether the MSCs contain *race conditions*. A race condition can result from the fact that we have only a limited control on the order between pairs of events that include at least one receive event (except for two receives corresponding to messages sent from the same process, according to the fifo semantics). For example, the MSC in Figure 1 contains two receive

events of process $P1$ (of messages $M5$ and $M6$). Since each process line is one dimensional, the MSC notation forces choosing one of the receive events to appear above the other. However, these two messages were sent from different processes, $P2$ and $P3$, and it might happen that $M6$ arrives quicker than $M5$. Thus, there is no reason to trust that these messages will arrive in the particular order depicted using the MSC.

Formally, we can define a race condition for pairs of MSC *receive* events $p, q \in V$ for messages sent from different processes such that $L(p) = L(q)$, i.e., p and q appear on the same process line. A race occurs if $p < q$, i.e., p appears above q on the process line, and it is *not* the case that $p <^* q$, i.e., there is no path from p to q according to the relation $<$. Detecting races in an MSC is thus simple. All we need is to calculate the transitive closure $<^*$ and compare it against relation $<$.

It is shown in [1] that the calculation of the transitive closure $<^*$ of $<$ is quadratic in the number of events, and not cubic as is the general case for transitive closure. This stem from the fact that the number of *immediate* successors of each event p under $<$ (i.e., events q such that $p < q$, and there is no r such that $p < r < q$) if limited to 2.

We can define the race conditions for HMSCs. This turns out to be an undecidable problem [11]. We regain decidability by limiting the structure of the HMSCs, as described in Section 5.

Acknowledgement

I would like to thank my long time collaborator in studying message sequence charts Anca Muscholl for many years of fruitful and pleasant collaboration.

References

[1] R. Alur, G. H. Holzmann, and D. A. Peled. An analyzer for message sequence charts. *Software Concepts and Tools*, 17(2):70–77, 1996.

[2] R. alur and D. Peled and W. Penczek, Model Checking of Causality Properties, Logic in Computer Science 1995, San Diego, CA, 90-100.

[3] R. Alur and M. Yannakakis. Model checking of message sequence charts. In *Proc. of CONCUR'99*, LNCS 1664, 114–129, 1999.

[4] E. Gunter, A. Muscholl, D. Peled, Compositional Message Sequence Charts, Tools and Algorithms for the Construction and Analysis of Systems (TACAS) 2001, LNCS 2031, 496–511.

[5] G. Holzmann, *Design and Validation of Computer Protocols*, Prentice-Hall, 1992.

[6] ITU-T Recommendation Z.120, Message Sequence Chart (MSC), 1996.

[7] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 1994.

[8] P. Madhusudan. Reasoning about sequential and branching behaviours of message sequence graphs. In *Proc. of ICALP'01*, LNCS 2076, 809–820, 2001.

[9] A. Mazurkiewicz, Basic notions of trace theory, REX workshop 1988, 285–363, LNCS 354, Springer, 1988.

[10] A. Muscholl and D. Peled. Message sequence graphs and decision problems on Mazurkiewicz In *Proc. MFCS'99*, LNCS 1672, 81–91, 1999.

[11] A. Muscholl, D. Peled, and Z. Su. Deciding properties of message sequence charts. In *Proc. of FoSSaCS'98*, LNCS 1378, 226–242, 1998.

[12] E. Ochmanski, Recognizable trace languages, *The Book of Traces*, V. Diekert, G. Rozenberg, (eds.), World Scientific, 1995, 167–204.

[13] D. Peled. Specification and verification of message sequence charts. In *Proc. of FORTE/PSTV 2000*, 2000.

[14] M. Y. Vardi, P. Wolper, An automata-theoretic approach to automatic program verification, Proceedings of the 1st Annual Symposium on Logic in Computer Science IEEE, 1986, 332–344.